Coding Club

Cognitive and Systematic Musicology Laboratory,

Ohio State University

Written by Hubert Léveillé Gauvin

leveillegauvin.1@osu.edu

Contents

1 Preface

1 Conventions Used in this Document

Code enclosed in a in solid-line box represents scripts, while code not enclosed in a box should be entered directly at the terminal. New lines are indicated by a small Arabic number in the left margin. If a line of code is not preceded by an Arabic number, it is not a new line, but rather a prolongation of the previous line (i.e. the line was wrapped to fit the pdf, but should be entered as one line at the terminal).

2 Known Bugs

At the moment, some lines of code won't work if you simply copy and paste them to the terminal (I need to figure out why). If this happens, try to manually type the code, or paste it in a text file and delete the extra spaces generated by copying it.

2 Using APIs

3 Who's in Space?

To use an API, we're going to use the `curl` function. In its most basic form, `curl` needs an URL to interact with something. Let's use `curl` to see who's in space right now using the following API: http://api.open-notify.org/astros.json

```
curl "http://api.open-notify.org/astros.json"
```

Yay! `curl` has a bunch of options. One of the useful one is `-s`, which enables silent mode and hides the progress bar.

```
curl -s "http://api.open-notify.org/astros.json"
```

`JSON` is a text format that is ideal data-interchange language. It is human readable, but as you can see, it can be a bit overwhelming. This is where `jq` comes handy. `jq` is a command-line tool that allows you to parse `JSON` file. Unfortunately, it is not preinstalled on your machine. To add it, I recommend you used `brew`. The `brew` package manager is like an app-store for the terminal. You can install `brew` on your machine following these instructions (OS X: https://brew.sh/ ; Ubuntu: http://linuxbrew.sh/).

> **Note: I haven't tried the Ubuntu version.**

Once `brew` is installed, you can easily add new tools to your machine. To add `jq`:

```
brew install jq
```

> **Note: For more information about `jq`, visit:**
> https://stedolan.github.io/jq/manual/v1.5/

Let's see what `jq` does:

```
curl -s "http://api.open-notify.org/astros.json" | jq
```

`JSON` files are organized as name/value pairs. For example, to know how many people are in space, we can call the name "number":

```
curl -s "http://api.open-notify.org/astros.json" | jq '.number'
```

`jq` follows the `UNIX` philosophy, meaning that it can interact with other commands. Above, we saw that it can receive input from another command. We can also send its output elsewhere. For example, let's use the stream-editor `sed` to convert "6" to "six":

```
curl -s "http://api.open-notify.org/astros.json" | jq '.number' | sed 's/6/six/g'
```

Just like with any other UNIX tools, we can assign the above command to a variable and call that variable later:

```
number_of_astronauts_in_space=$(curl -s "http://api.open-notify.org/astros.json" | jq '.number' | sed 's/6/six/g')
echo "There are $number_of_astronauts_in_space astronauts in space"
```

Now, imagine we want to create a list with the name of everyone in space right now. Let's try with something like this:

```
curl -s "http://api.open-notify.org/astros.json" | jq '.people.name'
```

That didn't work! jq gives us the following error message:

```
jq: error (at <stdin>:0): Cannot index array with string "name".
```

This is because the values stored in "people" are stored as an array. Arrays are represented by square brackets in JSON. To access arrays with jq, we can use the following syntax:

```
curl -s "http://api.open-notify.org/astros.json" | jq '.people[].name'
```

This is useful, but it would be even better if we could create a CSV file with the name of the astronaut and their craft. Let's try something like this:

```
curl -s "http://api.open-notify.org/astros.json" | jq '.people[].name, .people[].craft'
```

This gave us a list of all the people followed by a list of all the crafts. But we want to have name then craft for everyone in space. Let's re-work our query:

```
curl -s "http://api.open-notify.org/astros.json" | jq '.people[] | .name, .craft'
```

Better. Notice how were are using pipes within jq. We know we're still in jq because we haven't closed that single quote yet. This is still a list, but at least it's in the right order. Thankfully for us, jq has a built-in function to create CSV file called @csv. It does, however, require that we organize our data has an array, we can do that using square brackets:

```
curl -s "http://api.open-notify.org/astros.json" | jq '.people[] | [.name, .craft] | @csv'
```

This worked, but it's pretty messy. This is because mintinlinebashjq is adding extra " " to make sure things are separated properly. But since our original data was already enclosed in " ", we don't really need mintinlinebashjq to do this. We can specify that we want the output to be "raw" using the mintinlinebash-r option:

```
curl -s "http://api.open-notify.org/astros.json" | jq -r '.people[] | [.name, .craft] | @csv'
```

We can make our output even nicer by adding a header.

```
curl -s "http://api.open-notify.org/astros.json" | jq -r
'["NAME", "CRAFT"], (.people[] | [.name, .craft]) | @csv'
```

We did it! Since this is UNIX, we can redirect our output to a CSV file if we want to:

```
curl -s "http://api.open-notify.org/astros.json" | jq -r
'["NAME", "CRAFT"], (.people[] | [.name, .craft]) | @csv' > whosinspace.csv
```

The CSV file we just created can be used with other softwares, like R for example. But sometimes we just want to look at the data in the terminal. `csvkit` is a useful toolkit that allows you to view and manipulate CSV file directly in the terminal. If you have mintinlinebashbrew on your machine, you can install `csvkit` easily:

```
brew install csvkit
```

One of the useful tool that is included in the `csvkit` is `csvlook`, which displays CSV files in a nice table:

```
curl -s "http://api.open-notify.org/astros.json" | jq -r
'["NAME", "CRAFT"], (.people[] | [.name, .craft]) | @csv' | csvlook
```

Of course, `csvlook` can also open local files:

```
csvlook whosinspace.csv
```

4 Grabbing the Weather

The first API we used was pretty simple. It did not require any ID nor offered options. It really only did one thing: return information about who's in space. But most of the time, APIs will be more sophisticated. In this exercise, we'll be using the OpenWeatherMap API to fetch information about the weather. First, we'll need to sign up for an API key at: http://openweathermap.org/appid

After signing-up online, you'll receive your API key by email. Since everyone's API key is different, let's assign our API ID to a variable:

```
my_id=6ff3595d244317ecf2a4a17976e7XXXX
```

Note: You'll need to replace 6ff3595d244317ecf2a4a17976e7XXXX with your own ID.

We can now print our key anytime we need it:

```
echo $my_id
```

Note: This variable will be available as long as your session is running. If you close your terminal, your variable will disappear.

We're ready to use make our first request. Let's start by looking for the current weather in Columbus:

```
curl -s "https://api.openweathermap.org/data/2.5/weather?q=columbus&APPID=$my_id" | jq
```

That's a lot of information! Let's see... There is something called `main.temp`, but the value associated with it seems very high. Let's look at the API documentation online: http://openweathermap.org/current. Ah! Let's change the default unit to Fahrenheit. We can do this by modifying the API endpoint (i.e. then end of our url) to specify our preferred unit:

```
curl -s "https://api.openweathermap.org/data/2.5/weather?q=columbus&units=imperial&APPID=$my_id" |
jq
```

Just like we created a variable to store our ID, we can create a variable called `$city` to store the name of the city we're interested in:

```
city="columbus"
curl -s "https://api.openweathermap.org/data/2.5/weather?q=$city&units=imperial&APPID=$my_id" | jq
```

Now, imagine we're interested in finding out what the weather is like in Paris, we can simply update our variable `$city` and keep the same query:

```
city="paris"
curl -s "https://api.openweathermap.org/data/2.5/weather?q=$city&units=imperial&APPID=$my_id" | jq
```

We now know what the weather is like in Paris, France. But what if we were interested in knowing the weather in Paris, Ontario, "the Prettiest Little Town in Canada"? According to the online documentation, cities can be specified by city names (as we did previously), but also using either ISO 3166 country codes, city ID, geographic coordinates, or ZIP codes. Let's use city ID. Open Weather Map has a `JSON` file with all the cities available and their unique ID. You can download that file from the following url: http://bulk.openweathermap.org/sample/city.list.json.gz. We can also use `curl` to download the file, and `gunzip` to extract it:

```
curl -L "http://bulk.openweathermap.org/sample/city.list.json.gz" > "city.list.json.gz"
gunzip -kv "city.list.json.gz"
```

Note: `curl` **can be pretty slow.**

Let's look at the file we just downloaded. Since this is a pretty big file, we'll use the `head` command to only display the first 50 lines:

```
head -n 50 "city.list.json"
```

Since this is a `JSON` file, we might want to use `jq` command to display it in a nice way. Let's try it:

```
head -n 50 "city.list.json" | jq
```

That didn't work! That's because `JSON` is a hierarchical format, and by using the head command, we destroyed its hiercharchy. However, if we were to use `jq` on the whole file, things would work normally. Now since `city.list.json` is a pretty big file, this will take a couple of seconds. We can calculate how long it will take by adding `time` before the command you want to measure:

```
time jq '.' "city.list.json"
```

That took just a little bit more than 29 seconds on my machine! But it worked as expected. Now that we have a list of all the cities and their respective IDs, we need to find Paris, Ontario. We have a couple of ways to do this. Let's have a look at our `JSON` file. Right above the city name is its ID, and right below the city name is its country. This means that we could use `grep` to search for the city we're looking for, use the `-B1` option to tell `grep` to print one extra line before each match, and the `-A1` to print one extra line after our match:

```
grep -B1 -A1 "Paris" "city.list.json"
```

That worked out okay. We can be even more specific by filtering our output file. Let's see how many cities in Canada are named "Paris." We'll use `grep` to search for the line `"country": "CA"` and specify that, this time, we want to print to extra lines before our match:

```
grep -A1 -B1 "Paris" "city.list.json" | grep -B2 '"country": "CA"'
```

So we found the ID we we're looking for using `grep`. Alternatively, `jq` has a built-in search function:

```
jq '.[] | select(.name | match("paris";"i")) | select(.country | match("CA";"i"))' "city.list.json"
```

Let's breakdown our last query. We asked `jq` to open the array using `[]`, search within `.name` for the case-insensitive (`"i"`) string `"paris"`, then within those results, search within `.country` for a case-insensitive string matching `"CA"`, the country code for Canada. Since we're really interested in the city ID, let's be even more specific:

```
jq '.[] | select(.name | match("paris";"i")) | select(.country | match("CA";"i")) | .id'
"city.list.json"
```

Awesome. We can now copy and paste this ID somewhere. But wouldn't it be better to assign it to a variable? In `BASH`, you can assign the outcome of a command to a variable using the following method:

```
city_id=$(jq '.[] | select(.name | match("paris";"i")) | select(.country | match("CA";"i")) | .id'
"city.list.json")
```

If you've been following closely, you might wonder why we hard coded the string `"paris"` in the previous query, instead of using the shell variable `$city`. The answer is that shell variables cannot be used as is in a `jq` command. However, we can reassign a shell variables to `jq` variables. In the following command, we are using the `--arg` option to reassign the shell variable `$city` to the `jq` variable `$CITY`:

```
city_id=$(jq --arg CITY "$city"
'.[] | select(.name | match($CITY;"i")) | select(.country | match("CA";"i")) | .id'
"city.list.json")
```

Let's make sure it worked by using `echo` to print the value of `$city_id` to our terminal:

```
echo $city_id
```

Great! We can now use our variable `$city_id` within our API query. Since Canada uses the metric system, let's also change the system were a using:

```
curl -s "https://api.openweathermap.org/data/2.5/weather?id=$city_id&units=metric&APPID=$my_id" |
jq
```

Now, let's imagine that we are employees working for the city of Paris, Ontario, and that we want to write a small weather command that displays a one-sentence summary of the weather. Something that we could automatically tweet every morning for example. For example, it could be something like this:

> The weather is currently -3.67C in Paris, Ontario. Overcast clouds. Temperatures are expected to climb up to -3C.

To do this, we'll need to retrieve three bits of information: `.temp`; `.temp_max`; `.description`. Notice how for `.description`, we're removing the `" "` from the string using `sed`, and calling `python` to capitalize the string:

```
current_temp=$(curl -s "https://api.openweathermap.org/data/2.5/weather?id=$city_id
&units=metric&APPID=$my_id" | jq '.main.temp')
temp_max=$(curl -s "https://api.openweathermap.org/data/2.5/weather?id=$city_id&units=metric&APPID=
$my_id" | jq '.main.temp_max')
description=$(curl -s "https://api.openweathermap.org/data/2.5/weather?id=$city_id
&units=metric&APPID=$my_id" | jq '.weather[].description' | sed 's/"//g' | python -c
"print raw_input().capitalize()")
echo "The weather is currently "$current_temp"C in Paris, Ontario. "$description
". Temperatures are expected to climb up to "$temp_max"C."
```

Since Canada is a bilingual country, it would be nice if we could translate our last ouptut to French. Fortunately, the `trans` command from the `translate-shell` package uses the Google Translate API (and others) to do just that. We can install it using `brew`:

```
brew install translate-shell
```

We can see if it worked:

```
which -a trans
```

If you have the `humdrum toolkit` installed on your computer, you already have another command called `trans`. That will be problematic. We can overcome this problem by renaming our newly installed `trans` command to `translate` using an `alias`:

```
alias translate=/usr/local/bin/trans
```

This worked as a temporarily solution, but won't work after we close our terminal. To make it permanent, we can add the `alias` to our `bash_profile`:

```
echo "alias translate=/usr/local/bin/trans" >> ~/.bash_profile
source ~/.bash_profile
```

Note: Since `man` is an independent program, we can't do `man translate`, but `translate -h` will work.

We can test our newly renamed command. We'll used the `-to` option to specify the target language and `-b` to enable `brief` mode:

```
translate -to "French" -b 'Hello'
```

It works! We can now translate our weather tweet to French:

```
echo "The weather is currently "$current_temp"C in Paris, Ontario. "$description
". Temperatures are expected to climb up to "$temp_max"C." | translate  -to "French" -b
```

Note: The Google Translate engine is getting pretty good, but it's still not perfect. It works ok for this demo, but if we we're to implement an actual Twitter bot, we would probably want the text to be translated by a human.

5 Using the Spotify API

So far we have encountered APIs requiring no authentication, and APIs requiring a simple API key. But sometimes, in addition to having a unique key, APIs will require that you use an access token. While your key is permanent, your access token will only be valid for a limited period of time. The OAuth protocol that Spotify uses for their API works that way.

Note: Spotify has a Interactive API Web Console that allows you to use their API without having to code (but what's the fun in that?). You can find more about it at: https://developer.spotify.com/web-api/console/

To use the Spotify API, you'll first need to register on https://developer.spotify.com/. Use your regular Spotify username and password to create your developer app. The app can be called anything. You will be granted a Spotify developer client ID and client secret.

Our next step is to use our new credentials to get a temporary access token. An important coding skill is to be able to look online for existing bits of code. For example, let's see if we can find an existing script that will allow us to get a Spotify access token. Let's Google: "get spotify api access token script". One of the first results was this link: https://gist.github.com/ahallora/4aac6d048742d5de0e65, which had the following script:

```php
<?php

$client_id = '<insert your spotify app client id>';
$client_secret = '<insert your spotify app client secret>';

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL,            'https://accounts.spotify.com/api/token' );
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt($ch, CURLOPT_POST,           1 );
curl_setopt($ch, CURLOPT_POSTFIELDS,     'grant_type=client_credentials' );
curl_setopt($ch, CURLOPT_HTTPHEADER,     array('Authorization: Basic '.base64_encode($client_id
.':'.$client_secret)));

$result=curl_exec($ch);
echo $result;

?>
```

We are going to use this existing script to get an access token. First, we'll modify the script by adding our client id and secret:

```php
<?php

$client_id = '5cf8945daa8d4d2ca7862aef2bc4XXXX';
$client_secret = '69dae24b3fe44a6990171e5d7e78XXXX';

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL,            'https://accounts.spotify.com/api/token' );
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt($ch, CURLOPT_POST,           1 );
```

```
curl_setopt($ch, CURLOPT_POSTFIELDS,     'grant_type=client_credentials' );
curl_setopt($ch, CURLOPT_HTTPHEADER,     array('Authorization: Basic '.base64_encode($client_id
.':'.$client_secret)));

$result=curl_exec($ch);
echo $result;

?>
```

Second, we'll add a PHP she-bang line at the very beginning of our script to tell our shell that this is written in PHP:

```
#!/usr/bin/php
<?php

$client_id = '5cf8945daa8d4d2ca7862aef2bc4XXXX';
$client_secret = '69dae24b3fe44a6990171e5d7e78XXXX';

$ch = curl_init();
curl_setopt($ch, CURLOPT_URL,            'https://accounts.spotify.com/api/token' );
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt($ch, CURLOPT_POST,           1 );
curl_setopt($ch, CURLOPT_POSTFIELDS,     'grant_type=client_credentials' );
curl_setopt($ch, CURLOPT_HTTPHEADER,     array('Authorization: Basic '.base64_encode($client_id
.':'.$client_secret)));

$result=curl_exec($ch);
echo $result;

?>
```

We'll save this file as `spotifyToken.php`. Finally, our last step is to change the permissions associated with this script to make it executable:

```
chmod +x spotifyToken.php
```

Let's see what the script does:

```
./spotifyToken.php
```

By now, you should recognize that the data we received is in the `JSON` format. Let's use `jq` to make it look pretty:

```
./spotifyToken.php | jq .
```

This tells us what our token is ("`BQCU3...IMXXXX`"), its type ("`Bearer`"), and when it expires (in `3600` seconds, i.e. 1 hour). Since the token is so long, it might be useful to assign it to a variable. We'll use `jq` to print the value associated with `.access_token` and then `sed` to remove the `" "`:

```
token=$(./spotifyToken.php | jq '.access_token' | sed 's/"//g')
```

Note: This token will expire in 1 hour. If it expires, simply re-run the command above to get a new access token.

Let's see if it worked:

```
echo $token
```

Now that we have a valid token, we can make our first query. Spotify's API documentation is very detailed and can be found here: https://developer.spotify.com/web-api/endpoint-reference/ (and information about audio analysis can be found here: https://web.archive.org/web/20160528174915/http://developer.echonest.com/docs/v4/_static/AnalyzeDocumentation.pdf). In addition to our access token, each query will need 1) an access method, 2) an endpoint, and 3) a Spotify ID, (e.g. a track ID, album ID, and artist ID, etc).

1. The method of authentication is indicated on the Spotify website. For researchers, we will almost exclusively use "GET." GET is also the default method used by `curl`, so we won't have to worry about this too much.

2. Endpoints are used to indicate what type of information you want to retrieve. Simply copy the endpoint from the website and paste appropriately in the URL. For example: `/v1/albums/{id}/tracks`

Note: All endpoints should begin with a forward slash (/). On the Spotify website, the forward slash for "Audio Analysis for a Track" endpoint is missing. You will need to manually add the forward slash at the beginning of this endpoint to access this information.

3. Spotify IDs can be found though the Spotify app. Simply select the track, album, or artist, click share, and select URI to copy the ID.

Note: By default, if you copy a Spotify ID from the app, it will come in this format: spotify:track:4BRkPBUxOYffM2QXVlq7aC. To use this ID with the Spotify API, you will need to trim the beginning part and only keep 4BRkPBUxOYffM2QXVlq7aC. We can do this using the following `sed` command: `echo "spotify:track:4BRkPBUxOYffM2QXVlq7aC" | sed 's/.*://g'`.

Note: Some endpoints require two types of IDs. For example, to get access to a user's specific playlist, one would need `user_id` and `playlist_id`.

Okay, let's try to have a list of all the songs on the Beatles' album "Revolver" (Spotify ID: 3PRoXYsngSwjEQWR5PsHWR):

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR" -H "Authorization: Bearer $token
" | jq .
```

By skimming this file, it looks like the information we're looking for is under `.tracks`, then `.items`, then `.name`. Since `.item` is an array, we'll need to use `[]` brackets in our query:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer
$token" | jq '.tracks.items[].name'
```

Sometimes, it can be hard to get around `JSON` files, especially if they are very long. The following command (which I found on an online forum), although very long and convoluted, can be really useful to get a big picture of how one file is organized:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer
$token" | jq '[
path(..)
| map(
```

```
    if type == "number" then
        "[]"
    else
        tostring
    end
)
| join(".")
| split(".[]")
| join("[]")
]
| unique
| map("." + .)
| .[]'
```

Let's make this into an alias and add it to our `.bash\_profile` so we never have to type this monstrosity ever again. Open `~/.bash_profile` with your favourite text editor (I'll use `TextEdit` in this example):

```
open -a TextEdit ~/.bash_profile
```

Copy and paste the following command as the last line of you `~/.bash_profile`, then save and quit:

```
alias jq_overview='jq '"'"'[path(..) | map(if type == "number" then "[]" else tostring end)
| join(".") | split(".[]") | join("[]")] | unique | map("." + .) | .[]'"'"''
```

Finally run source `~/.bash_profile`:

```
source ~/.bash_profile
```

Ok, so we managed to find print a list of all the song. Let's make it a little bit more interesting by also retrieving the duration of each song, and make it into a CSV file:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer
$token" | jq -r '.tracks.items[] | [.name, .duration_ms] | @csv'
```

Now, imagine we wanted to make our CSV file more useful by adding a column for artist (`.tracks.items[].artists.name`) and for album name (`.name`). The `@csv` filter included with `jq` is designed to convert arrays into CSV. However, since the value for album name is not part of the `.item` array, we'll need to think outside the box a little bit. One solution would be to simply hardcode the name of the album in our query. For example:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer
$token" | jq -r '.tracks.items[] | [.artists[].name, "Revolver", .name, .duration_ms] | @csv'
```

That works ok, but the fact that it's hard coded makes it harder for us to reuse the code. A better solution would be to assign the value of the album's name stored under `.name` to a variable, and then call that variable when needed:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer
$token" | jq -r
'.name as $album_name | .tracks.items[] | [.artists[].name, $album_name, .name, .duration_ms] | @csv'
```

Note: In the line above, we created a `jq` variable, not a `BASH` variable. This explains why the syntax to create a variable is different than the one we saw previously.

By default, the `v1/albums/{id}` endpoint will return a maximum of 20 songs. We can push that limit to 50 using the `limit` option:

```
curl -s "https://api.spotify.com/v1/albums/3exqrnwvtUAEVCwar8xIcs/tracks?limit=50" -H
"Authorization: Bearer $token" | jq '.items[].name'
```

Some albums, however, have more than 50 tracks. In order to get around this limitation, we will need to make more than one query. For example, we can get tracks 51-100 using a combination of `limit` and `offset`:

```
curl -s "https://api.spotify.com/v1/albums/3exqrnwvtUAEVCwar8xIcs/tracks?offset=50&limit=50" -H
"Authorization: Bearer $token" | jq '.items[].name'
```

Let's see what else we can do with the Spotify API. One of the endpoint is called "audio analysis" and gives us some information about single songs. Let's try it out:

```
curl -s "https://api.spotify.com/v1/audio-analysis/2vEQ9zBiwbAVXzS2SOxodY" -H
"Authorization: Bearer $token" | jq '.'
```

Again, since this is a big file, let's use the `jq_overview` alias we created above to have a quick overview of the file's organization:

```
curl -s "https://api.spotify.com/v1/audio-analysis/2vEQ9zBiwbAVXzS2SOxodY" -H
"Authorization: Bearer $token" | jq_overview
```

One of the keys is called `.bars[].start`. We can use this to estimate how many measures are in a song. First, let's start by having a look at the data:

```
curl -s "https://api.spotify.com/v1/audio-analysis/2vEQ9zBiwbAVXzS2SOxodY" -H
"Authorization: Bearer $token" | jq '.bars[].start'
```

This gives us a series of timestamps representing the estimated beginning of each measure. We can count how many measures there are by using the `wc -l` command:

```
curl -s "https://api.spotify.com/v1/audio-analysis/2vEQ9zBiwbAVXzS2SOxodY" -H
"Authorization: Bearer $token" | jq '.bars[].start' | wc -l
```

We can also use the Spotify API to retrieve information like the tempo of a song. We will do this using the "audio features" endpoint:

```
curl -s "https://api.spotify.com/v1/audio-features/2vEQ9zBiwbAVXzS2SOxodY" -H
"Authorization: Bearer $token" | jq '.tempo'
```

Now imagine we wanted to get the tempo of all the songs on the Beatles' album "Revolver." The audio features endpoint allows you to retrieve information for many songs at once, as long as your Spotify IDs are comma-separated. First, we'll use the album endpoint to retrieve the Spotify ID for all the songs on the album:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer
$token" | jq -r '.tracks.items[].uri'
```

Our next step is to clean our IDs to create a nice comma-separated list. First, let's remove the first part of the ID. We'll use `sed` to replace any string of character ending with a colon with nothing:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer $token" | jq -r '.tracks.items[].uri' |  sed 's/.*://g'
```

Next, we'll need to convert the newlines (represented by `\n`) to commas. We'll use `tr` to do that:

```
curl -s "https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/" -H "Authorization: Bearer $token" | jq -r '.tracks.items[].uri' |  sed 's/.*://g' | tr "\n" ","
```

Let's assign this list of IDs to a variable:

```
list_IDs=$(curl -s https://api.spotify.com/v1/albums/3PRoXYsngSwjEQWR5PsHWR/ -H "Authorization: Bearer $token" | jq -r '.tracks.items[].uri' |  sed 's/.*://g' | tr "\n" ",")
```

Great. Now we can go back to the audio features endpoint and retrieve the tempo information for all those Spotify IDs:

```
curl -s "https://api.spotify.com/v1/audio-features?ids=$list_IDs" -H "Authorization: Bearer $token" | jq '.audio_features[].tempo'
```

We can do the same thing for mode (where 1 is major and 0 is minor):

```
curl -s "https://api.spotify.com/v1/audio-features?ids=$list_IDs" -H "Authorization: Bearer $token" | jq '.audio_features[].mode'
```

And of course, we can create a CSV file that combines both:

```
curl -s "https://api.spotify.com/v1/audio-features?ids=$list_IDs" -H "Authorization: Bearer $token" | jq -r '.audio_features[] | [.mode, .tempo] | @csv'
```